307307
Part 1 – Introduction to Generative AI

Topics

- Fundamentals of neural networks
- Evolution from single Perceptrons to MLPs
- Detailed MLP architecture (input, hidden, and output layers)
- Mathematical representations
- Various activation functions (Sigmoid, ReLU, etc.)
- Backpropagation and training methodologies
- Loss functions and optimization techniques
- Architecture design considerations
- Real-world applications
- Advantages and limitations
- Modern MLP variants and implementations

What is a Neural Network?

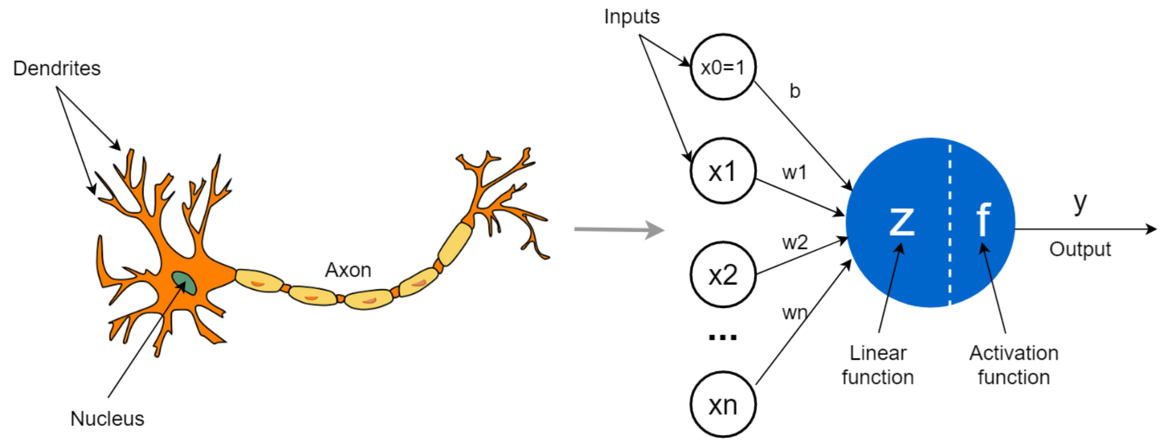
- To understand how LLMs work, it helps to understand some fundamentals of neural networks, as they are the foundation of many AI systems.
- Neural networks are inspired by the brain and consist of interconnected “neurons” that adjust connection strengths during learning.
- 1943 – McCulloch & Pitts: Proposed the first mathematical model of a neuron (MCP neuron), a binary threshold logic gate.
- They showed that networks of such neurons could compute any logical function.
- This laid the theoretical foundation for artificial neural networks.

Warren Sturgis McCulloch

(1898 – 1969)

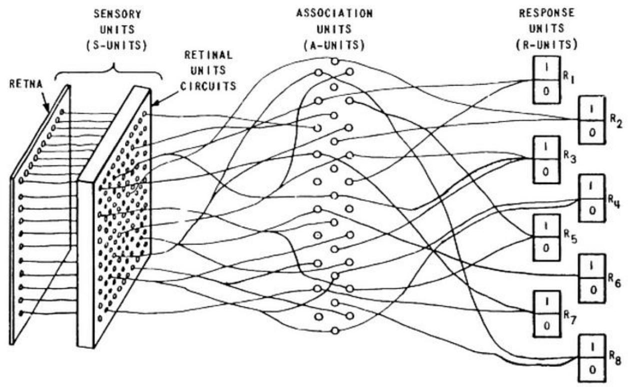
Walter Harry Pitts, Jr.

(1923 – 1969)

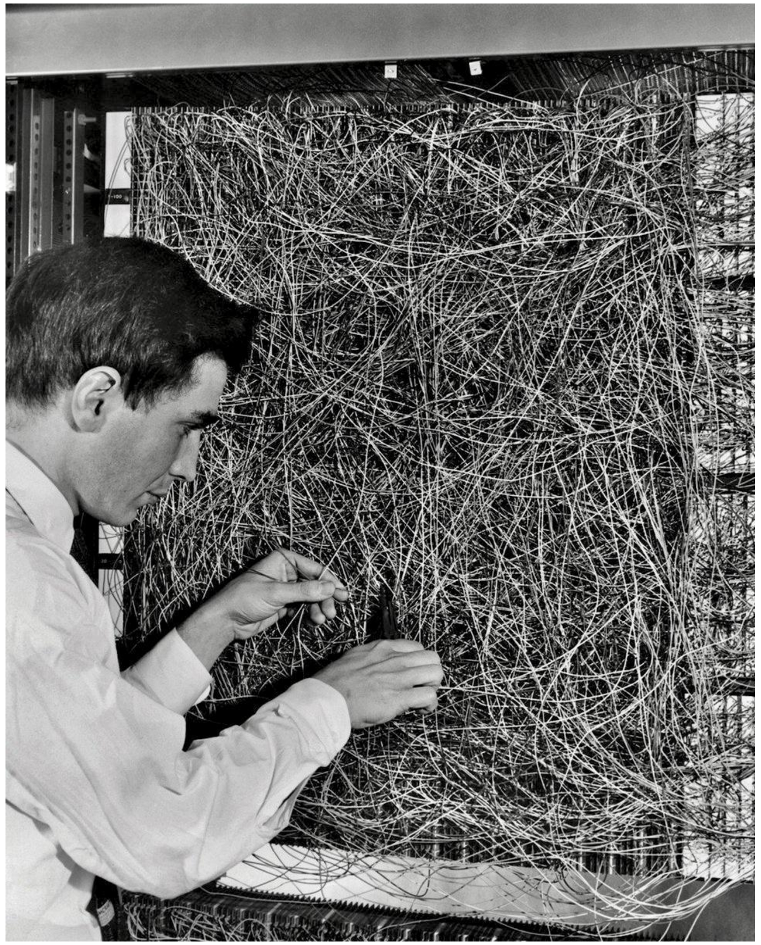


The Perceptron: Building Block of Neural Networks

- 1958 – Frank Rosenblatt (inspired by McCulloch & Pitts): Introduced the **Perceptron** (the Mark I Perceptron at Cornell).
- Designed to recognize patterns in visual data.
- It was Implemented in hardware.
- The perceptron is a binary classifier: input → weighted sum → threshold → output.
- Rosenblatt also described multi-layer perceptrons, but at the time no algorithm existed to train them effectively.



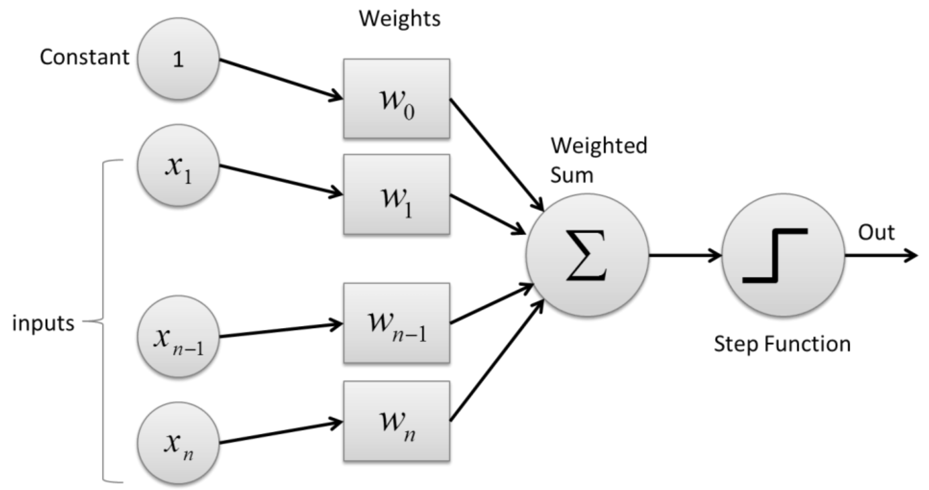
F. Rosenblatt



The diagram shows **Rosenblatt's Perceptron (1958)**, which is essentially an early form of a **multi-layer perceptron (MLP)**: sensory units (S-units) connected to association units (A-units), which in turn connect to response units (R-units). Inspired by the visual cortex, it could learn to classify input patterns by adjusting connection weights. While the theory was purely mathematical, Rosenblatt also built an **electromechanical implementation called the Mark I Perceptron**, which used an array of photocells as the retina, analog circuits for weighted connections, and motors to adjust the weights physically. This made it one of the first tangible demonstrations of machine learning hardware.

The Perceptron

- Inputs: $x_1, x_2, ..., x_n$
- Weights: $w_1, w_2, ..., w_n$
- Bias: b
- Activation function: Step function
- Output: 1 if weighted sum > threshold, 0 otherwise

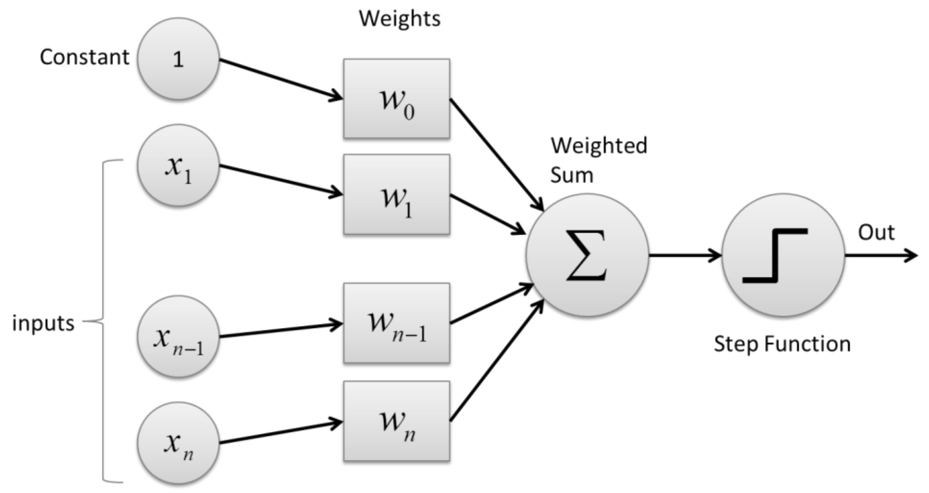


How a Perceptron Works

1. Multiply each input by its corresponding weight
2. Sum all weighted inputs
3. Add the bias term
4. Apply the activation function
5. Output the result

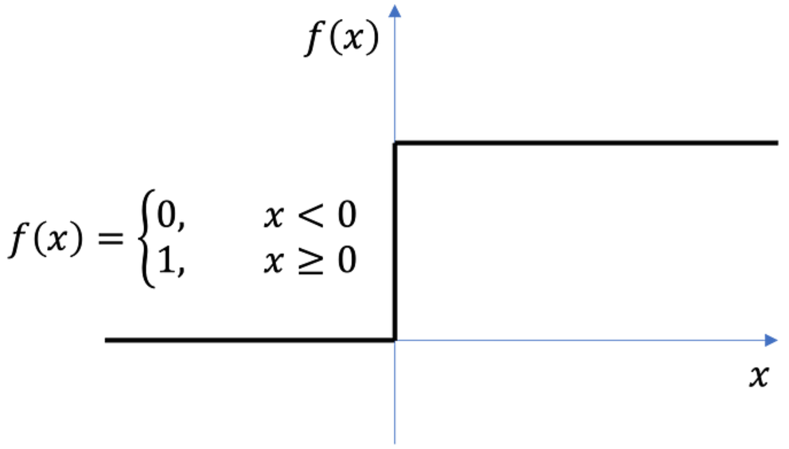
Mathematically:

- $z = w_1x_1 + w_2x_2 + \ldots + w_nx_n + b$
- output = activation(z)



Perceptron Activation Function

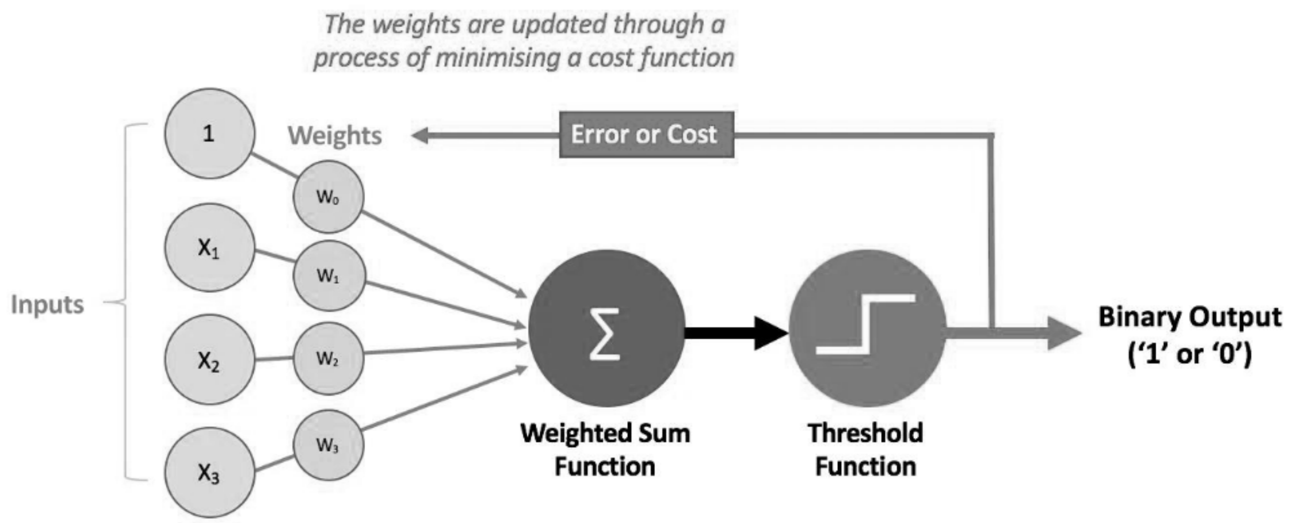
- **Step Function**:
  - Output: 1 if z ≥ 0, 0 if z < 0
  - Used in original perceptrons
  - Not differentiable at 0



How Perceptron Learn

For each training example:

1. Calculate predicted output y_pred
2. Calculate error: error = y_true - y_pred
3. Update weights: w_new = w_old + learning_rate * error * x
4. Update bias: b_new = b_old + learning_rate * error



Step-by-Step Hand Calculation for AND Gate

Let's work through the perceptron learning algorithm by hand and train the Percetron to solve the gate:

- Learning rate (η) = 0.1
- Initial weights (randomly assigned): $w_1 = 0.3$, $w_2 = -0.1$
- Initial bias: $b = 0.2$

First Iteration:

Record # 1: Inputs: $x_1 = 0$, $x_2 = 0$, output = 0

- Weighted sum: $z = w_1x_1 + w_2x_2 + b = 0.3(0) + (-0.1)(0) + 0.2 = 0.2$
- Activation: output = 1 (since $z > 0$)
- True output: $y = 0$
- Error: error = y - output = 0 - 1 = -1
- Weight updates:
  - $w_1 = w_1 + \eta * \text{error} * x_1 = 0.3 + 0.1 * (-1) * 0 = 0.3$
  - $w_2 = w_2 + \eta * \text{error} * x_2 = -0.1 + 0.1 * (-1) * 0 = -0.1$
  - $b = b + \eta * \text{error} = 0.2 + 0.1 * (-1) = 0.1$

Training data

A (Input 1)	B (Input 2)	X = (A.B)
0	0	0
0	1	0
1	0	0
1	1	1

Step-by-Step Hand Calculation for AND Gate

Record # 2: Inputs: $x_1 = 0$, $x_2 = 1$, output = 0

- Weighted sum: $z = w_1x_1 + w_2x_2 + b = 0.3(0) + (-0.1)(1) + 0.1 = 0$
- Activation: output = 1 (since $z \geq 0$)
- True output: $y = 0$
- Error: error = y - output = 0 - 1 = -1
- Weight updates:
  - $w_1 = w_1 + \eta * \text{error} * x_1 = 0.3 + 0.1 * (-1) * 0 = 0.3$
  - $w_2 = w_2 + \eta * \text{error} * x_2 = -0.1 + 0.1 * (-1) * 1 = -0.2$
  - $b = b + \eta * \text{error} = 0.1 + 0.1 * (-1) = 0$

Record # 3: Inputs: $x_1 = 1$, $x_2 = 0$, output = 0

- Weighted sum: $z = w_1x_1 + w_2x_2 + b = 0.3(1) + (-0.2)(0) + 0 = 0.3$
- Activation: output = 1 (since $z > 0$)
- True output: $y = 0$
- Error: error = y - output = 0 - 1 = -1
- Weight updates:
  - $w_1 = w_1 + \eta * \text{error} * x_1 = 0.3 + 0.1 * (-1) * 1 = 0.2$
  - $w_2 = w_2 + \eta * \text{error} * x_2 = -0.2 + 0.1 * (-1) * 0 = -0.2$
  - $b = b + \eta * \text{error} = 0 + 0.1 * (-1) = -0.1$

Step-by-Step Hand Calculation for AND Gate

Record # 4: Inputs: $x_1 = 1$, $x_2 = 1$, output = 1

- Weighted sum: $z = w_1x_1 + w_2x_2 + b = 0.2(1) + (-0.2)(1) + (-0.1) = -0.1$
- Activation: output = 0 (since $z < 0$)
- True output: $y = 1$
- Error: error = y - output = 1 - 0 = 1
- Weight updates:
    - $w_1 = w_1 + \eta * \text{error} * x_1 = 0.2 + 0.1 * 1 * 1 = 0.3$
    - $w_2 = w_2 + \eta * \text{error} * x_2 = -0.2 + 0.1 * 1 * 1 = -0.1$
    - $b = b + \eta * \text{error} = -0.1 + 0.1 * 1 = 0$

End of Iteration 1:

- Updated weights: $w_1 = 0.3$, $w_2 = -0.1$
- Updated bias: $b = 0$

Second Iteration

Record # 1: Inputs: $x_1 = 0$, $x_2 = 0$, Output = 0

- Weighted sum: $z = w_1x_1 + w_2x_2 + b = 0.3(0) + (-0.1)(0) + 0 = 0$
- Activation: output = 1 (since $z \geq 0$)
- True output: $y = 0$
- Error: error = y - output = 0 - 1 = -1
- Weight updates:
  - $w_1 = w_1 + \eta * \text{error} * x_1 = 0.3 + 0.1 * (-1) * 0 = 0.3$
  - $w_2 = w_2 + \eta * \text{error} * x_2 = -0.1 + 0.1 * (-1) * 0 = -0.1$
  - $b = b + \eta * \text{error} = 0 + 0.1 * (-1) = -0.1$

Example 2: Inputs: $x_1 = 0$, $x_2 = 1$, output = 0

- Weighted sum: $z = w_1x_1 + w_2x_2 + b = 0.3(0) + (-0.1)(1) + (-0.1) = -0.2$
- Activation: output = 0 (since $z < 0$)
- True output: $y = 0$
- Error: error = y - output = 0 - 0 = 0
- Weight updates (no change as error = 0):
  - $w_1 = 0.3$
  - $w_2 = -0.1$
  - $b = -0.1$
- **After several iterations**, the perceptron will converge to weights that correctly classify all AND gate examples.

Python Implementation

```python
from sklearn.linear_model import Perceptron
import numpy as np

  # Training data for AND gate
X = np.array([[0, 0], [0, 1], [1, 0], [1, 1]])
y = np.array([0, 0, 0, 1])

  # Initialize and train Perceptron
model = Perceptron(max_iter=100, eta0=0.1,
  random_state=42)
model.fit(X, y)
# Results
print("Weights:", model.coef_)
print("Bias:", model.intercept_)
print("Predictions:", model.predict(X))
```

```
Weights: [[0.2 0.2]]
Bias: [-0.2]
Predictions: [0 0 0 1]
```

The code shows a scikit-learn Perceptron implementation for the AND gate problem.

The code:

1. Imports NumPy, scikit-learn's Perceptron, and matplotlib
2. Sets up the training data for the AND gate
3. Initializes a Perceptron with 100 max iterations and a random seed of 42
4. Trains the perceptron on the AND gate data
5. Prints the learned weights, bias, and predictions

The output shows:

- **Weights: [[0.2 0.2]]** - The perceptron learned to assign a weight of 0.2 to both inputs
- **Bias: [-0.2]** - The bias is -0.2
- **Predictions: [0 0 0 1]** - The perceptron correctly classified all four examples of the AND gate

With these weights and bias, the decision function is: 0.2×(input1) + 0.2×(input2) – 0.2

For the four input combinations:

- [0,0]: $0.2\times0 + 0.2\times0 - 0.2 = -0.2 < 0 \rightarrow$ output 0
- [0,1]: $0.2\times0 + 0.2\times1 - 0.2 = 0 \rightarrow$ output 0
- [1,0]: $0.2\times1 + 0.2\times0 - 0.2 = 0 \rightarrow$ output 0
- [1,1]: $0.2\times1 + 0.2\times1 - 0.2 = 0.2 > 0 \rightarrow$ output 1

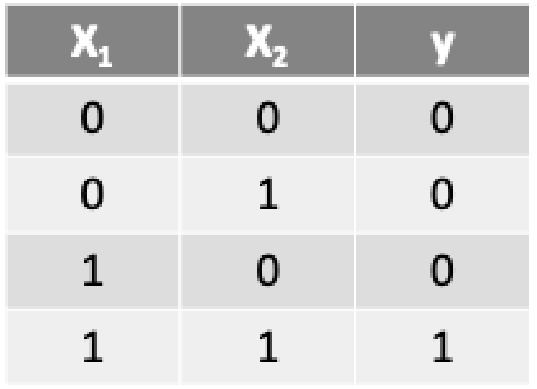
This perceptron implements the AND gate logic.

The decision boundary is the line $2x_1 + 2x_2 - 0.2 = 0$, which separates the point (1,1) from the other three points.

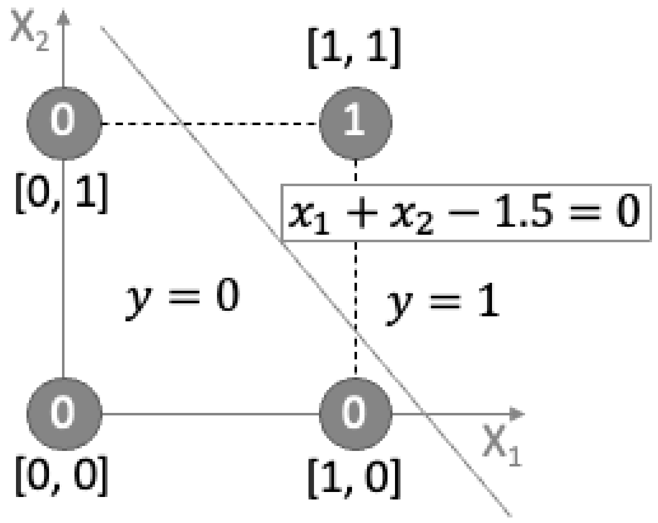
Open in Colab

Decision Boundary

- The perceptron learns a decision boundary: $w_1x_1 + w_2x_2 + b = 0$
- Points above the line are classified as 1
- Points below the line are classified as 0
- For AND gate, only the point (1,1) should be above the line

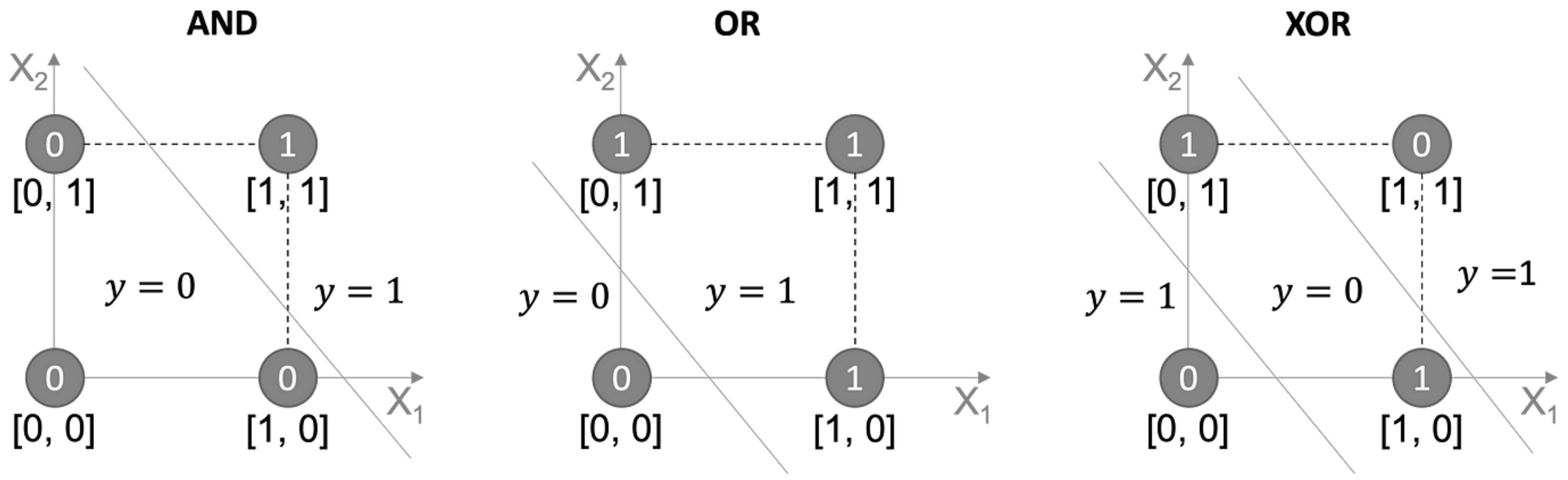


X_1	X_2	y
0	0	0
0	1	0
1	0	0
1	1	1



Limitations of Single Layer Perceptron

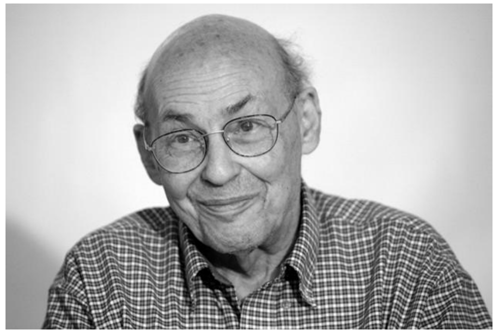
- Single layer perceptron can only learn linearly separable patterns
- It cannot solve XOR problem (need multiple layers)
- Simple update rule isn't suitable for complex problems
- There was no suitable algorithm to train a multi-layer perceptron.



The Multi-Layer Percecptron (MLP)

Limitations of the Perceptron

While useful for linearly separable problems, the single perceptron cannot solve complex problems like XOR classification, as demonstrated by Minsky and Papert in their 1969 book "Perceptrons.“



The Multi-Layer Perceptron

The Multi-Layer Perceptron addresses the limitations of the single perceptron by introducing:

- Multiple layers of neurons
- Non-linear activation functions
- More sophisticated learning algorithms

Structure of an MLP

Definition: An MLP is a class of feedforward artificial neural network that consists of at least three layers of nodes: **input**, **hidden**, and **output** layers.

Key Feature: Each neuron in one layer is connected to every neuron in the next layer (fully connected).

1. Input Layer

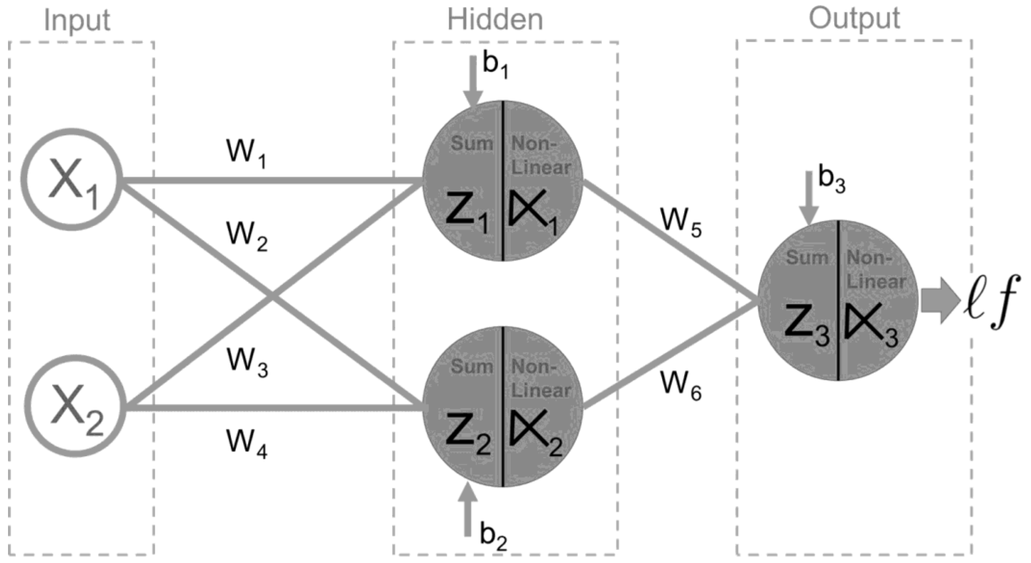
- Receives the raw input features
- One neuron per input feature
- No computation occurs here; inputs are simply passed forward

2. Hidden Layer(s)

- One or more layers between input and output
- Each neuron in a hidden layer:
- Receives inputs from all neurons in the previous layer
- Computes a weighted sum
- Applies a non-linear activation function
- Passes the result to the next layer

3. Output Layer

- Produces the final prediction or classification
- Structure depends on the task:
    - Regression: Often a single neuron with linear activation
    - Binary classification: One neuron with sigmoid activation
    - Multi-class classification: Multiple neurons (one per class) with softmax activation



Structure of an MLP

3. Neurons and Connections

- Each neuron computes a weighted sum of inputs and applies an activation function
- Fully connected between layers (dense connections)

4. Activation Functions

- Introduce non-linearity
- Examples: ReLU, Sigmoid, Tanh, Softmax

5. Loss Function

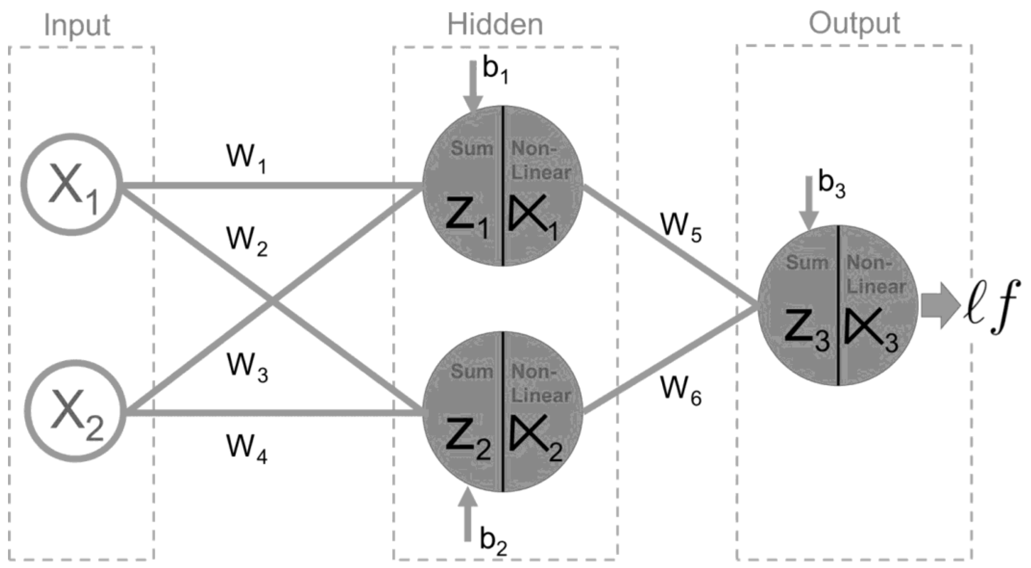
- Measures the error between predicted and true outputs
- Examples: Mean Squared Error, Cross-Entropy

6. Optimizer

- Updates weights to minimize loss
- Examples: SGD, Adam

7. Training Data

- Labeled data used to train the network
- Split into training, validation, and test sets



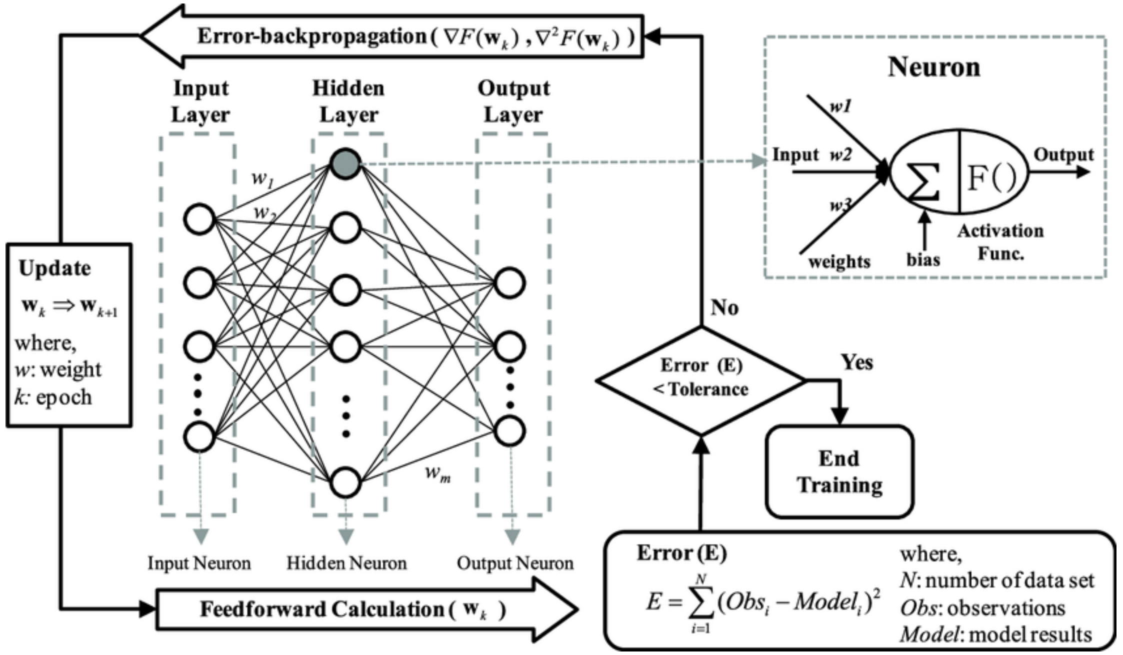
How an MLP Learn – Step by Step

Forward Propagation

- Input features pass through the network layer by layer.
- Each neuron computes a weighted sum of inputs and applies an activation function.
- The final layer produces a prediction.

Loss Computation

- A loss function measures the difference between predicted and actual outputs.
- Common loss functions:
    - Mean Squared Error (regression)
    - Cross-Entropy Loss (classification)



How an MLP Learn – Training Process

Backpropagation

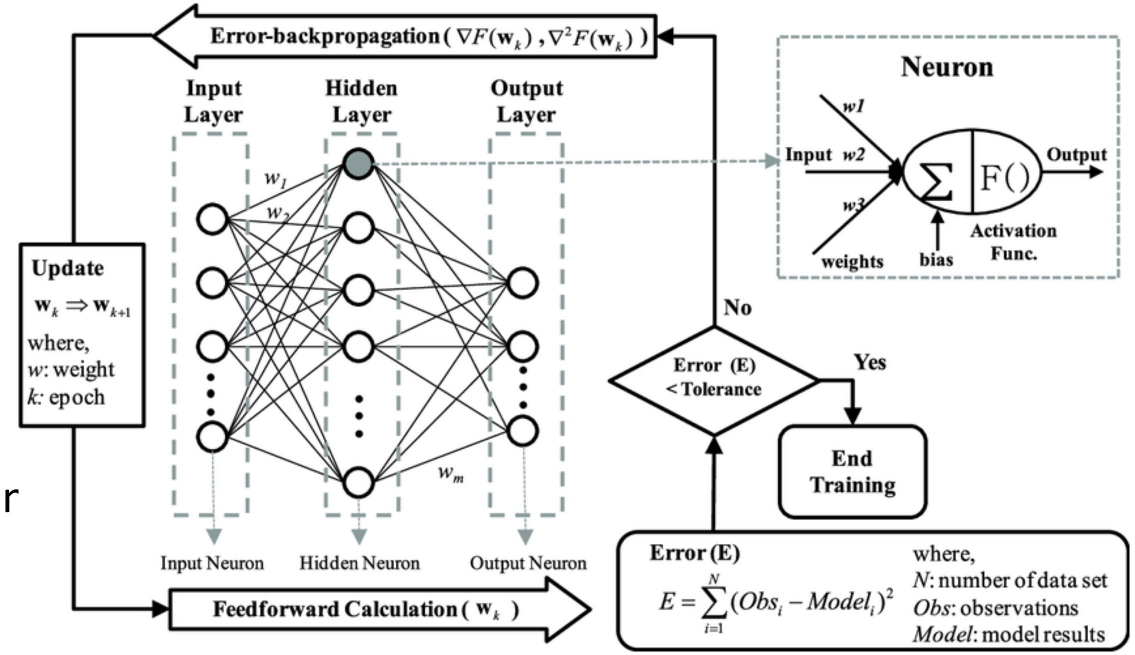
- Gradients of the loss are calculated with respect to each weight using the chain rule.
- This identifies how each weight contributed to the error.

Weight Update

- An optimizer (e.g., SGD) adjusts weights to reduce loss:
  - w := w - learning_rate × gradient
- This process repeats over multiple iterations (epochs) usir training data.

Goal

- Gradually minimize the loss and improve prediction accuracy.
- **Loss Function**: MSE for regression, Cross-Entropy for classification.
- **Optimization**: Backpropagation + Gradient Descent (or Adam).



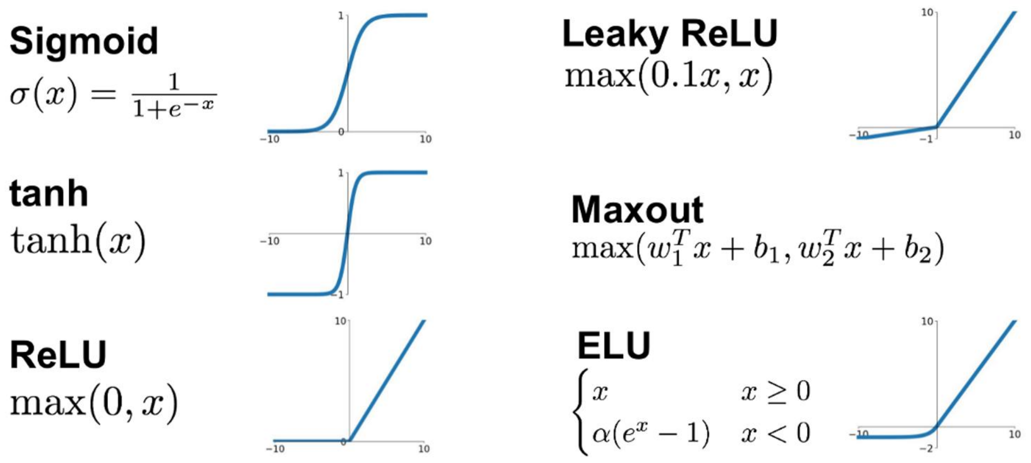
Activation Functions other than Step Function

- Neural Networks use activation functions other than the simple step function in the Perceptron.
- Activation Function helps the neural network use important information while suppressing irrelevant data points (i.e., allows local “gating” of information).

Common Activation Functions: A Toolbox for Neural Networks

- **Sigmoid**: Compresses input to a range between 0 and 1.
- **Tanh**: Like sigmoid but ranges from -1 to 1.
- **ReLU**: Passes positive values, zeroes out negatives.
- **Leaky ReLU**: Allows a small, non-zero gradient for negative inputs.
- **Maxout**: Chooses the most active linear response.
- **ELU**: Smoothly transitions through zero and allows negative outputs.

Each function offers a trade-off between simplicity, flexibility, and computational cost—choose based on the task and depth of your model



https://ml-explained.com/blog/activation-functions-explained

Institute of Analytics
Recognized Programme

University of Petra

Representing Weights as Matrices

Matrix Representations for Weights

1. **Weight Matrices**:

$W^{(1)}$: Weights from Input to Hidden Layer (shape: input_size × hidden_size):

$[\ w_{1,1}^{(1)}\ w_{1,2}^{(1)}\ ]$
$[\ w_{2,1}^{(1)}\ w_{2,2}^{(1)}]$

Hidden Layer Bias (b_1):
$[\ b_{11}\ b_{12}\ ]$

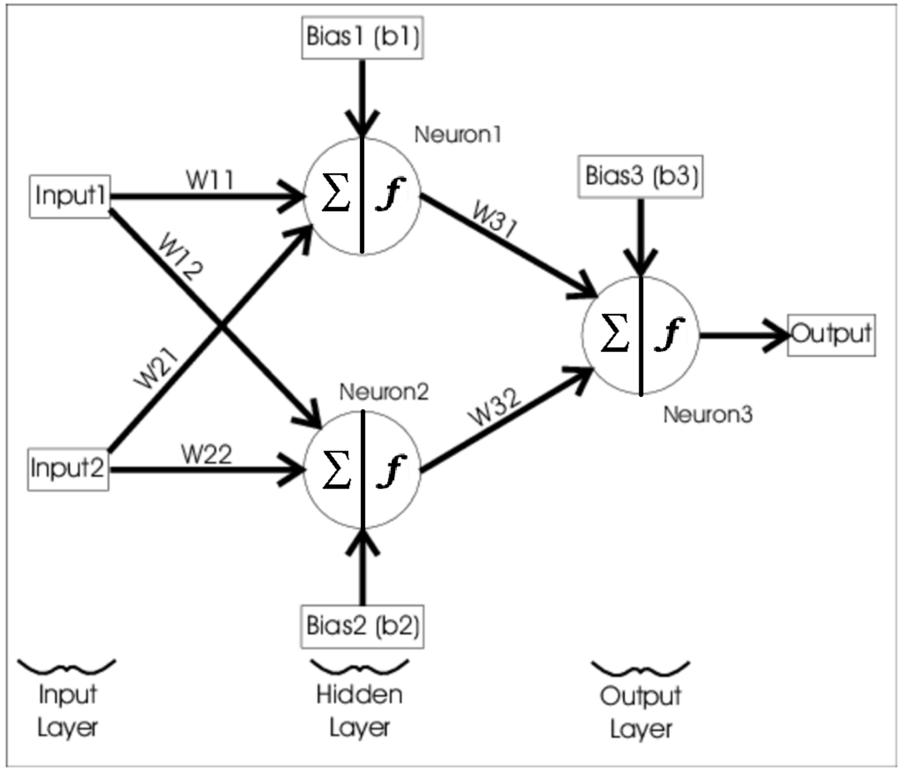
$W^{(2)}$: Weights from Hidden to Output Layer (shape: hidden_size × output_size):

$[\ w_{1,1}^{(2)}\ ]$
$[\ w_{2,1}^{(2)}\ ]$

Output Layer Bias (b_2):
$[\ b_{21}\ ]$

2. **Forward Pass Matrix Operations**:
   - Input to hidden layer: $Z_1 = X \cdot W_1 + b_1$
   - Hidden to output layer: $Z_2 = A_1 \cdot W_2 + b_2$

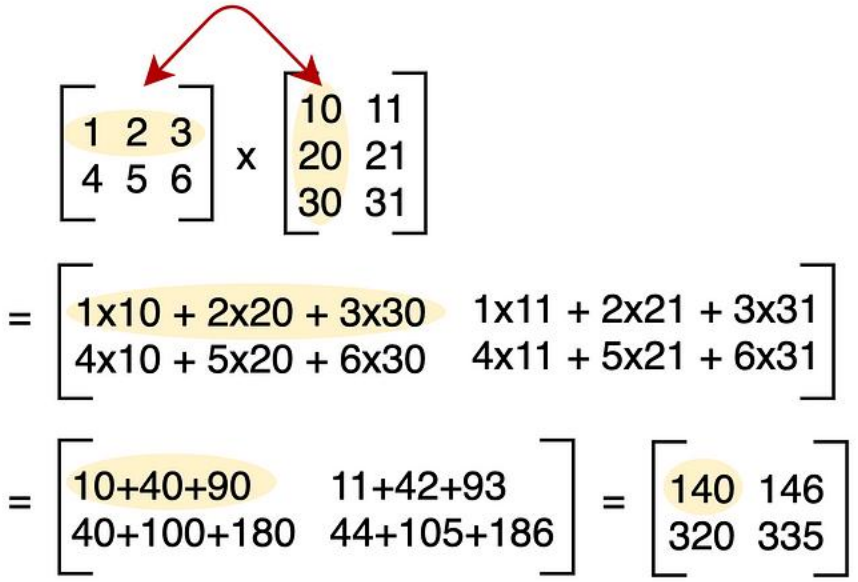
3.**Backpropagation Matrix Operations**:
   - Weight updates use matrix multiplication between layer activations and error gradients
   - W_2 update: $W_2 \mathrel{+}= A_1^T \cdot \text{delta}_2 \times \text{learning\_rate}$
   - W_1 update: $W_1 \mathrel{+}= X^T \cdot \text{delta}_1 \times \text{learning\_rate}$



The Neural Network Model to solve the XOR Logic (from: https://stopsmokingaids.me/)

NO ACTION REQUIRED

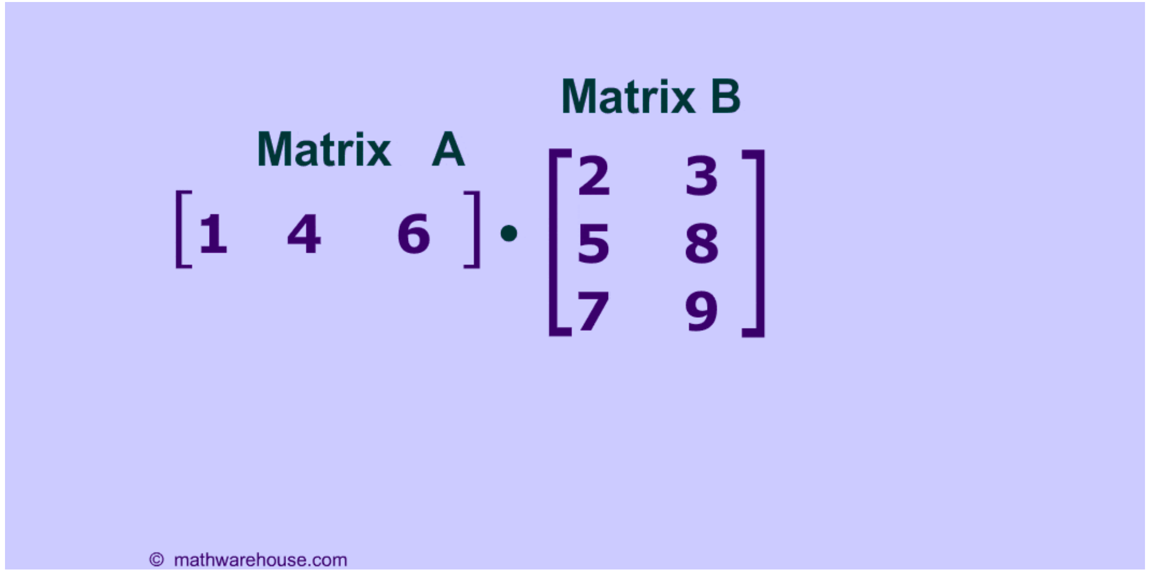
Matrix Multiplication



$$\begin{bmatrix}1 & 2 & 3\\4 & 5 & 6\end{bmatrix} \times \begin{bmatrix}10 & 11\\20 & 21\\30 & 31\end{bmatrix}$$

$$= \begin{bmatrix}1\times10 + 2\times20 + 3\times30 & 1\times11 + 2\times21 + 3\times31\\4\times10 + 5\times20 + 6\times30 & 4\times11 + 5\times21 + 6\times31\end{bmatrix}$$

$$= \begin{bmatrix}10+40+90 & 11+42+93\\40+100+180 & 44+105+186\end{bmatrix} = \begin{bmatrix}140 & 146\\320 & 335\end{bmatrix}$$



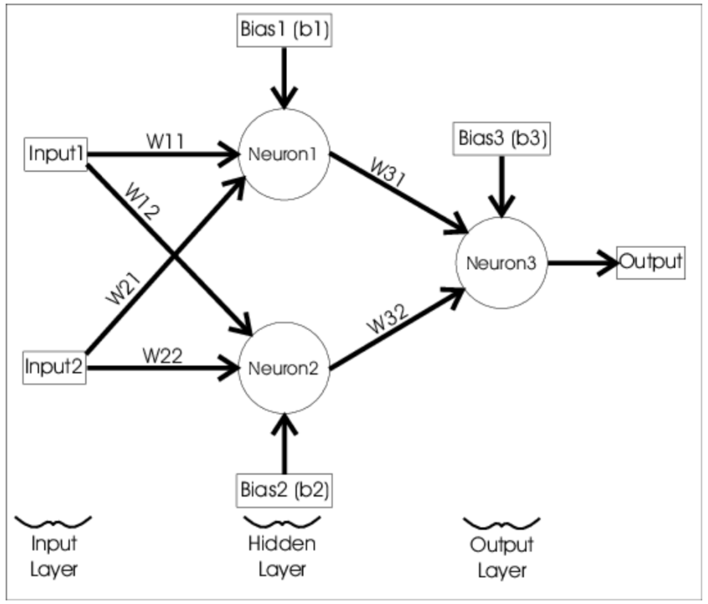
$$\begin{bmatrix}1 & 4 & 6\end{bmatrix} \cdot \begin{bmatrix}2 & 3\\5 & 8\\7 & 9\end{bmatrix}$$

Multi-Layer Perceptron in Python

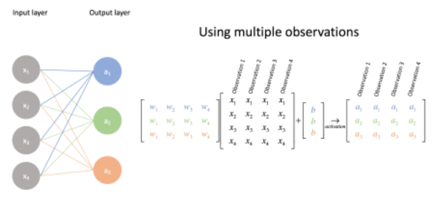
```python
from sklearn.neural_network import MLPClassifier
import numpy as np
# XOR input and output
X = np.array([[0, 0], [0, 1], [1, 0], [1, 1]])
y = np.array([0, 1, 1, 0])
# Define MLP with 1 hidden layer of 2 neurons (minimal config for XOR)
mlp = MLPClassifier(hidden_layer_sizes=(2,), activation='tanh’, solver='adam', learning_rate_init=0.01,
    max_iter=10000, random_state=42)
# Train the model
mlp.fit(X, y)
# Make predictions
predictions = mlp.predict(X)
print("Predictions:\n", predictions)
print("\nWeights (input to hidden):\n", mlp.coefs_[0])
print("\nBias hidden:\n", mlp.intercepts_[0])
print("\nWeights (hidden to output):\n", mlp.coefs_[1])
print("\nBias output:\n", mlp.intercepts_[1])
```

```
Weights (input to hidden):
 [[ 2.7144501   3.27401218]
 [-2.73418453 -3.17014048]]

Bias hidden:
 [ 1.21994174 -1.63451199]
```

```
Weights (hidden to output):
 [[-4.37775211]
 [ 4.46553876]]

Bias output:
 [3.61855675]
```



The Neural Network Model to solve the XOR Logic (from: https://stopsmokingaids.me/)



NO ACTION REQUIRED

Open in Colab

Institute of Analytics
Recognized Programme

Simple Neural Network in Python

```python
import numpy as np

# Simple feedforward neural network
def neural_network(x, weights):
    # First hidden layer with ReLU activation
    hidden = np.maximum(0, np.dot(x, weights[0]) + weights[1])  # ReLU activation
    # Output layer
    output = np.dot(hidden, weights[2]) + weights[3]
    return output

# Example network with random weights
input_size = 3
hidden_size = 4
output_size = 2

# Initialize random weights
W1 = np.random.randn(input_size, hidden_size)  # Input → Hidden
b1 = np.random.randn(hidden_size)              # Hidden bias
W2 = np.random.randn(hidden_size, output_size) # Hidden → Output
b2 = np.random.randn(output_size)              # Output bias
weights = [W1, b1, W2, b2]

# Example input
x = np.array([0.5, 0.3, 0.2])

# Forward pass
prediction = neural_network(x, weights)
print(f"Network prediction: {prediction}")
```

NO ACTION REQUIRED

Open in Colab

Backpropagation (Conceptual)

```python
import numpy as np

# Simplified backpropagation example
def train_step(x, y_true, weights, learning_rate=0.01):
    # Forward pass
    hidden = np.maximum(0, np.dot(x, weights[0]) + weights[1])  # ReLU
    y_pred = np.dot(hidden, weights[2]) + weights[3]

    # Compute loss (Mean Squared Error)
    loss = np.mean((y_pred - y_true)**2)

    # Backpropagation (simplified)
    # Output layer gradients
    grad_y_pred = 2 * (y_pred - y_true) / len(y_true)
    grad_W2 = np.dot(hidden.T, grad_y_pred)
    grad_b2 = np.sum(grad_y_pred, axis=0)

    # Hidden layer gradients
    grad_hidden = np.dot(grad_y_pred, weights[2].T)
    grad_hidden[hidden <= 0] = 0  # ReLU gradient
    grad_W1 = np.dot(x.T, grad_hidden)
    grad_b1 = np.sum(grad_hidden, axis=0)

       # Update weights
    weights[0] -= learning_rate * grad_W1
    weights[1] -= learning_rate * grad_b1
    weights[2] -= learning_rate * grad_W2
    weights[3] -= learning_rate * grad_b2

       return loss, weights

# Example usage
# (In practice, we would use frameworks like PyTorch or TensorFlow)
```

NO ACTION REQUIRED

Open in Colab

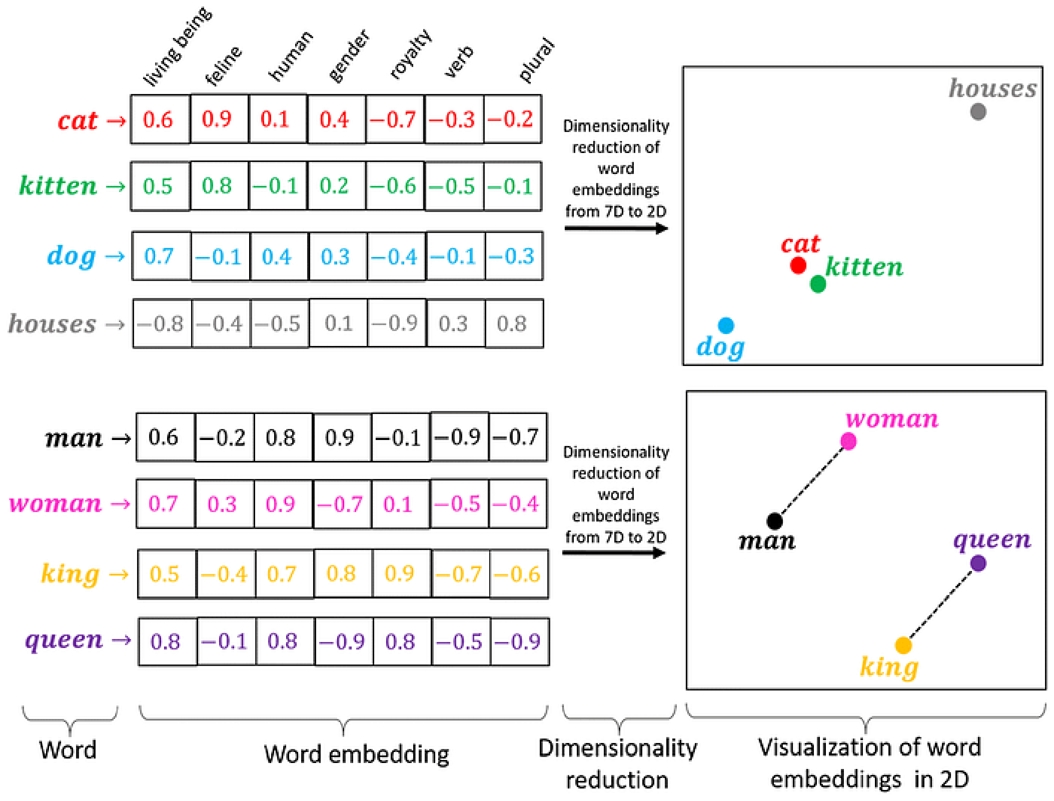
The Difference Between a Perceptron and an MLP?

Feature	Perceptron	Multilayer Perceptron (MLP)
Layers	Only 1 layer (no hidden layers)	2+ layers (has hidden layers)
Activation Function	Step function (hard threshold)	Nonlinear (e.g., ReLU, sigmoid, tanh)
Learning Rule	Simple rule: update on error	Gradient descent + backpropagation
Tasks It Can Solve	Only linearly separable problems	Nonlinear, complex problems

Introduction to Word Embeddings

Word Embeddings

- Word embeddings are numerical representations of words as vectors in a multi-dimensional space, used in Natural Language Processing (NLP) to capture semantic and syntactic relationships.
- By assigning a unique vector of continuous values to each word, these embeddings allow computers to process language by understanding that words with similar meanings have similar vectors.
- This enables machines to perform complex tasks like translation, sentiment analysis, and question answering more effectively than with simple representations like one-hot encoding.

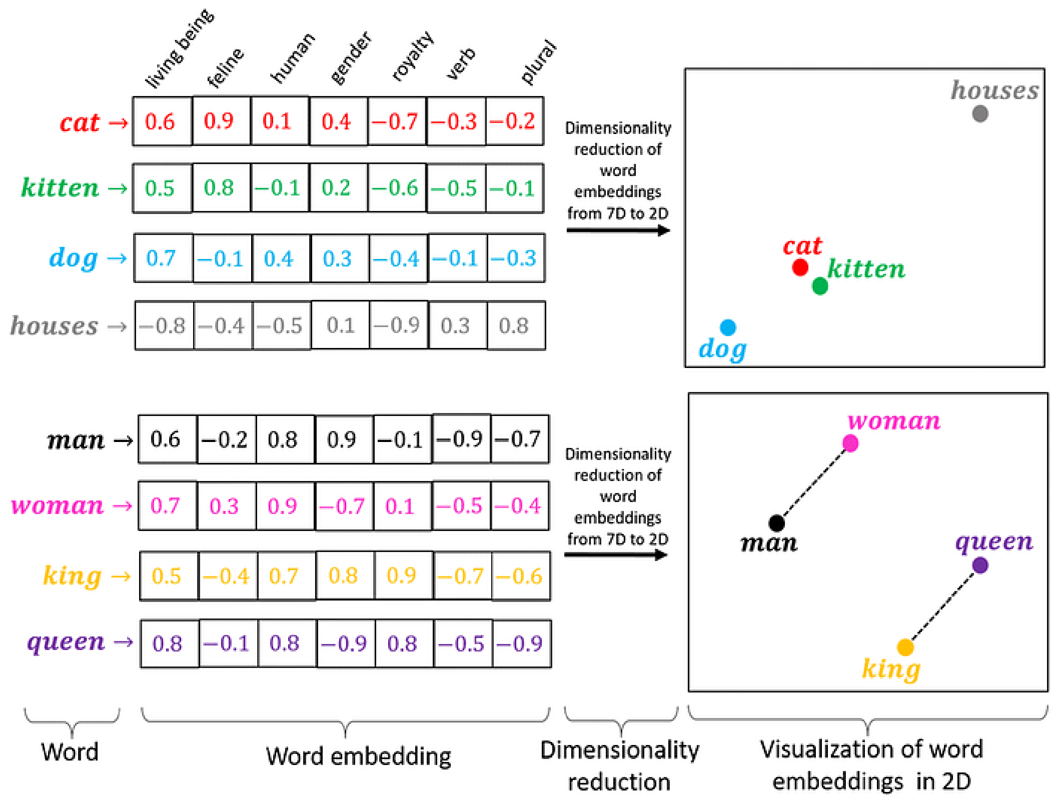


	living being	feline	human	gender	royalty	verb	plural
cat →	0.6	0.9	0.1	0.4	−0.7	−0.3	−0.2
kitten →	0.5	0.8	−0.1	0.2	−0.6	−0.5	−0.1
dog →	0.7	−0.1	0.4	0.3	−0.4	−0.1	−0.3
houses →	−0.8	−0.4	−0.5	0.1	−0.9	0.3	0.8
man →	0.6	−0.2	0.8	0.9	−0.1	−0.9	−0.7
woman →	0.7	0.3	0.9	−0.7	0.1	−0.5	−0.4
king →	0.5	−0.4	0.7	0.8	0.9	−0.7	−0.6
queen →	0.8	−0.1	0.8	−0.9	0.8	−0.5	−0.9

Dimensionality reduction of word embeddings from 7D to 2D
houses
cat
kitten
dog
Dimensionality reduction of word embeddings from 7D to 2D
woman
man
queen
king
Word
Word embedding
Dimensionality reduction
Visualization of word embeddings in 2D

Word Embeddings

- **Numerical representation**: Words are converted into vectors (lists of numbers) that can be processed by machine learning models.
- **Capturing relationships**: The position and direction of these vectors reflect the relationship between words. For example, the vectors for "king," "queen," and "man" would have a specific relationship that is similar to the relationship between "king," "woman," and "queen".
- **Semantic proximity**: Words with similar meanings, such as "lake" and "river," will have vectors that are closer to each other in the vector space.
- **Context-aware**: Modern embeddings can even provide different representations for the same word depending on its context (e.g., "bank" as a financial institution vs. the bank of a river).
- **Enabling NLP tasks**: These numerical representations are crucial for tasks like sentiment analysis, machine translation, text classification, and question answering.


Dimensionality reduction of word embeddings from 7D to 2D
houses
cat
kitten
dog
Dimensionality reduction of word embeddings from 7D to 2D
woman
man
queen
king
Word
Word embedding
Dimensionality reduction
Visualization of word embeddings in 2D

	living being	feline	human	gender	royalty	verb	plural
cat →	0.6	0.9	0.1	0.4	−0.7	−0.3	−0.2
kitten →	0.5	0.8	−0.1	0.2	−0.6	−0.5	−0.1
dog →	0.7	−0.1	0.4	0.3	−0.4	−0.1	−0.3
houses →	−0.8	−0.4	−0.5	0.1	−0.9	0.3	0.8

man →	0.6	−0.2	0.8	0.9	−0.1	−0.9	−0.7
woman →	0.7	0.3	0.9	−0.7	0.1	−0.5	−0.4
king →	0.5	−0.4	0.7	0.8	0.9	−0.7	−0.6
queen →	0.8	−0.1	0.8	−0.9	0.8	−0.5	−0.9

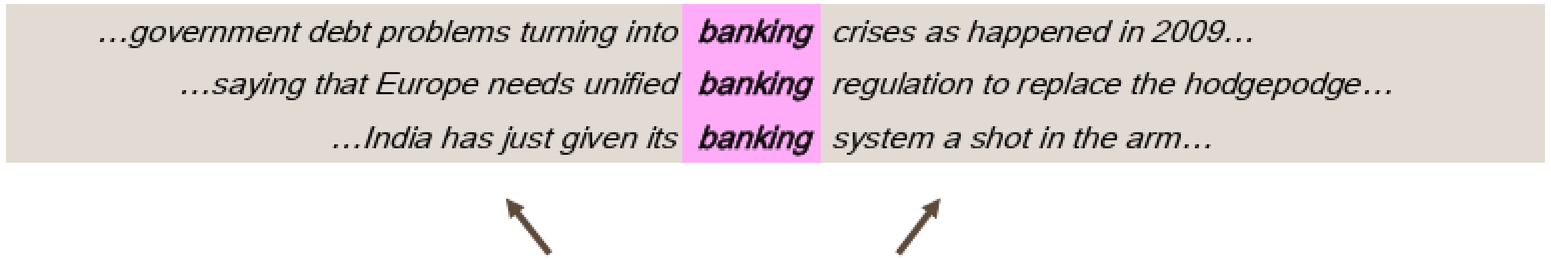
How Did We Represent Words Pre-2013

- Traditional models like Bag-of-Words (BoW) or TF-IDF, treat words as independent, ignoring semantic similarity.
- **One-hot encoding**: Sparse, binary vectors (dimension = vocabulary size)
- Example: "king" and "queen" are as unrelated as "king" and "banana" in BoW.

Word	Dimension 1 (cat)	Dimension 2 (dog)	Dimension 3 (fish)	Dimension 4 (bird)
cat	1	0	0	0
dog	0	1	0	0
fish	0	0	1	0
bird	0	0	0	1

The Evolution of Word Representations

- **Problem**: How do we represent meaning mathematically?
- **Solution**: Distributional hypothesis - "You shall know a word by the company it keeps" (J.R. Firth, 1957)
- J.R. Firth **did not** provide a detailed technical implementation like an algorithm or computational method. His statement was more of a **linguistic philosophy** or a **theoretical principle**, not a specific engineering method.
- And much later, it inspired the **distributional hypothesis** in computational linguistics, especially by scholars like Zellig Harris and later computational models (Word2Vec, etc.)



…government debt problems turning into ***banking*** *crises as happened in 2009…*

…saying that Europe needs unified ***banking*** *regulation to replace the hodgepodge…*

…India has just given its ***banking*** *system a shot in the arm…*

These context words will represent ***banking***

Word2Vec (Tomas Mikolov et al., 2013)

- Developed by Tomas Mikolov and team at Google.

Key Innovation

- Transformed NLP by creating dense vector representations through prediction-based models

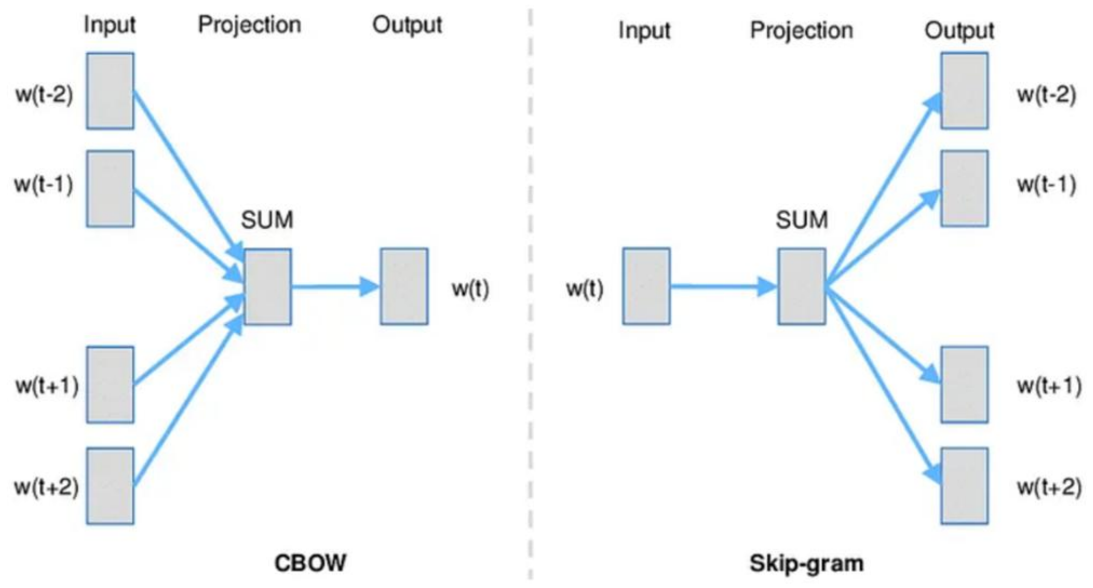
Two Architectures

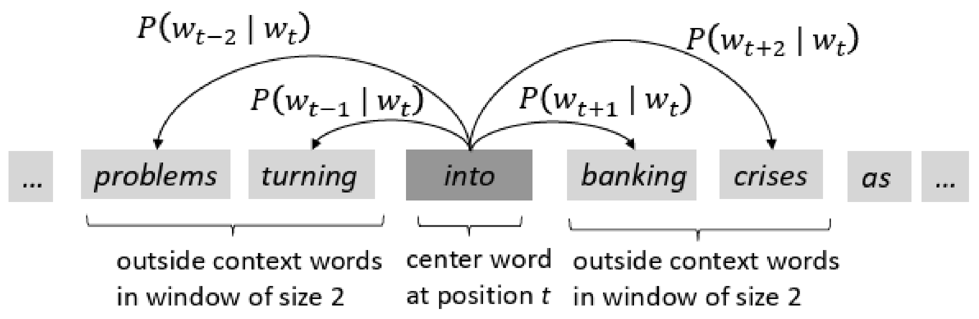
- **Continuous Bag of Words (CBOW)**:
    - Predicts target word from context words
    - Faster training, better for frequent words
- **Skip-gram**:
    - Predicts context words from target word
    - Better for rare words, captures more semantic information

Characteristics

- Uses **shallow neural networks** and trains on local context windows.
- Typically, 100-300 dimensions (vs. vocabulary size)
- Linear relationships: king - man + woman ≈ queen
- Efficient training through negative sampling
- Limitations: Fixed vectors, one vector per word regardless of context

CBow and Skip-Gram Models



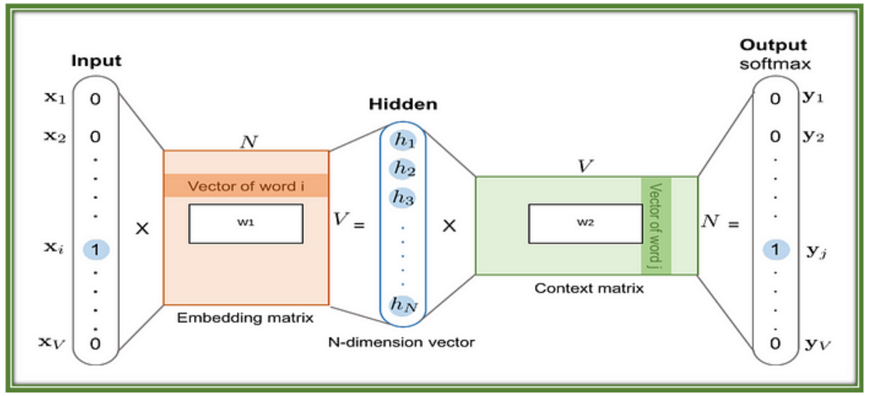


For each position t=1,…,T, predict context words within a window of fixed size m, given center word wt.

Skip-gram Architecture (Word2Vec)

This diagram illustrates how Word2Vec's **Skip-gram model** works:

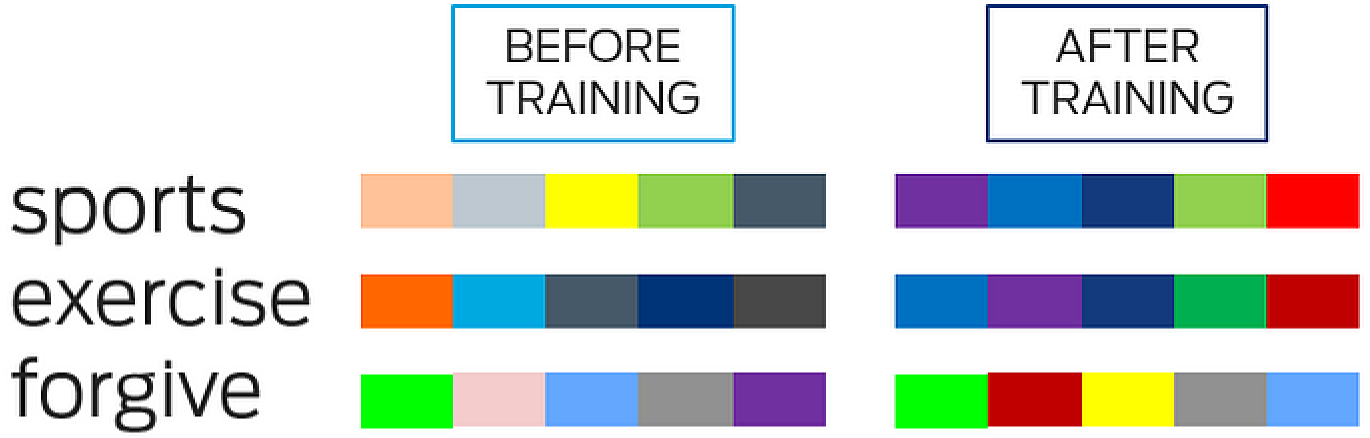
- **Input**: A one-hot encoded vector for the center word (word *i*).
- **Embedding Matrix**: Multiplies the input vector to produce a **dense embedding** (N-dimensional vector) — this becomes the **vector representation of the input word**.
- **Context Matrix**: The dense vector is then multiplied with another matrix to predict surrounding context words via softmax output.
- **Output**: A probability distribution over the vocabulary, aiming to maximize the likelihood of actual context words.
- This training process helps learn **meaningful word vectors** based on how words appear in context.



https://python.plainenglish.io/understanding-word-embeddings-tf-idf-word2vec-glove-fasttext-996a59c1a8d3

The Result of the Training Process - Similar Words -> Similar Vectors

- Visual example of how embedding changes before and after training. `sports` and `exercise` have similar embedding value post training because they are closely related

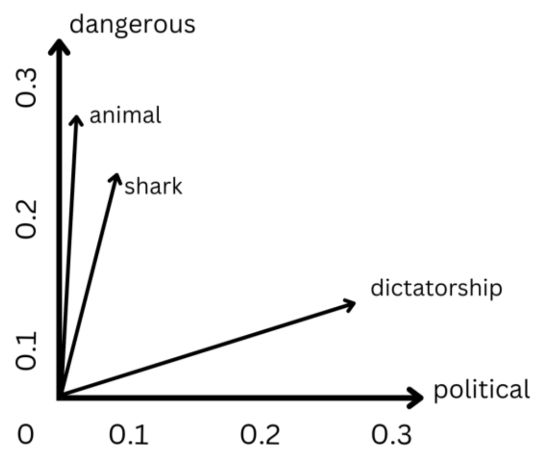


Vector Spaces and Word Embeddings

What is a Vector Space?

- A mathematical space where each word is represented as a point (or vector) in multi-dimensional space.
- Words are encoded as dense numerical vectors instead of one-hot or sparse representations (**Word Embeddings**) e.g., "king" → [0.21, 0.72,….,…., 0.35]
- Word Embeddings captures semantic and syntactic relationships about/between words.
- Each dimension potentially captures semantic meaning.
- These vectors are learned from text by models like Word2Vec or GloVe.

Word	Dimension 1 (political)	Dimension 2 (dangerous)
shark	0.05	0.22
animal	0.03	0.25
dangerous	0.07	0.32
political	0.31	0.04
dictatorship	0.28	0.15



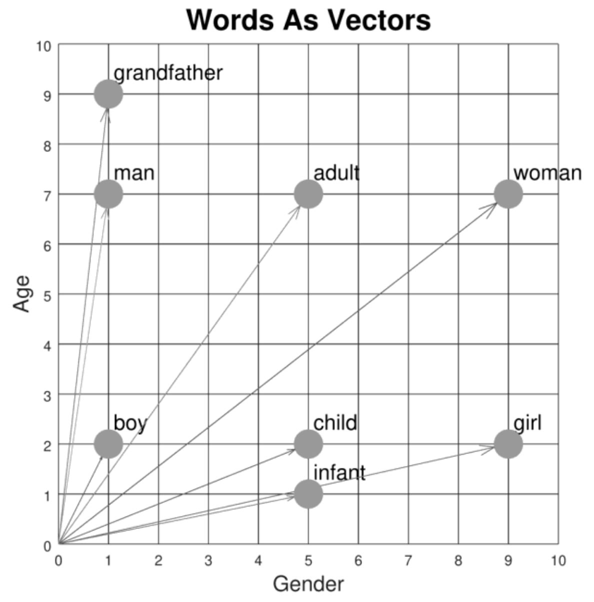
Vector Spaces and Word Embeddings

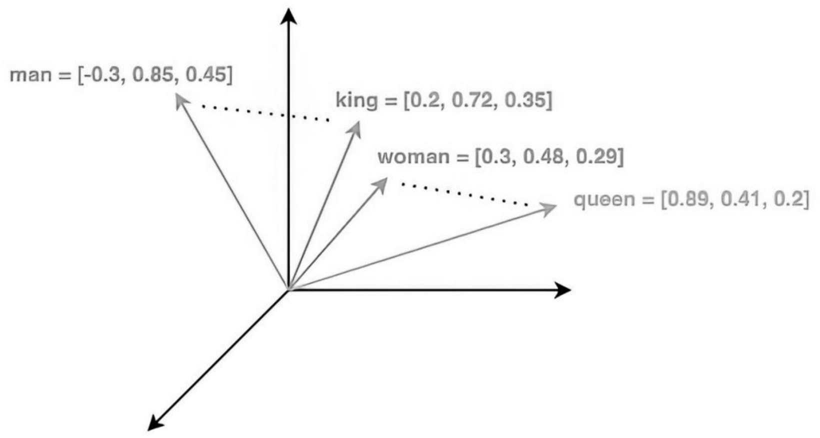
Why Use a Vector Space?

- Makes it possible to compare, visualize, and manipulate meanings of words using math.
- Enables operations like:
  - Similarity: "king" is close to "queen"
  - Analogy: "king" - "man" + "woman" ≈ "queen"

Properties of Vector Space

- Semantic relationships are preserved (e.g., "shark" is closer to "dangerous" than "political").
- Similar meanings → closer vectors.
- Dissimilar meanings → vectors farther apart.





Glove (Pennington, Socher, Manning 2014)

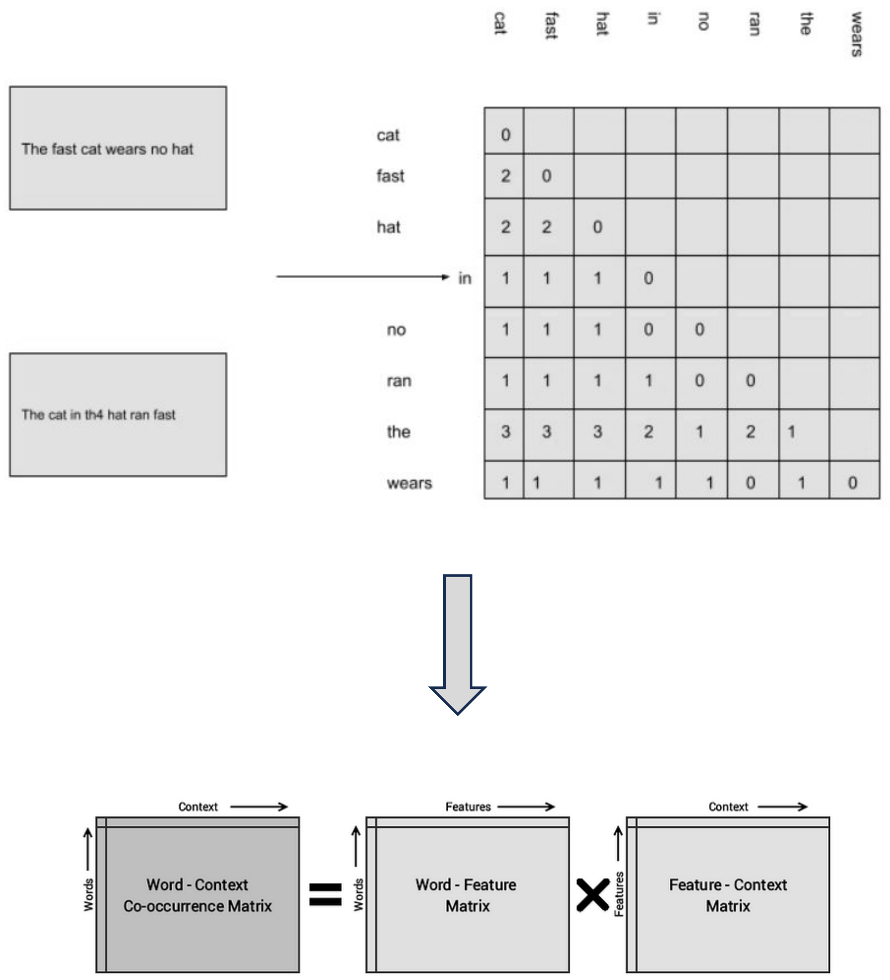
- Developed by Stanford NLP Group (Pennington, Socher, Manning)
- **Key Innovation**

Bridges the gap between count-based methods and prediction-based methods by using global co-occurrence statistics to learn word vectors

- **Approach**
- Builds a word-word co-occurrence matrix over a large corpus
- Learns embeddings by factorizing the matrix using a weighted least squares objective

Characteristics

- Captures global statistical information while maintaining useful properties of local context
- Produces dense word vectors (typically 100–300 dimensions)
- Linear relationships in vector space are preserved: king - man + woman ≈ queen
- Trained on massive corpora (Wikipedia, Common Crawl)
- **Limitations: Ignores context variability—still one vector per word regardless of usage**


The fast cat wears no hat
The cat in th4 hat ran fast
Context
Words
Word - Context Co-occurrence Matrix
=
Features
Words
Word - Feature Matrix
×
Context
Features
Feature - Context Matrix

	cat	fast	hat	in	no	ran	the	wears
cat	0							
fast	2	0						
hat	2	2	0					
in	1	1	1	0				
no	1	1	1	0	0			
ran	1	1	1	1	0	0		
the	3	3	3	2	1	2	1	
wears	1	1	1	1	1	0	1	0

Measuring Similarity Between Word Vectors

Why Compare Word Vectors?

- Word embeddings map words into a vector space.
- **Words with similar meanings** are placed **close together** in that space.
- To quantify this "closeness," we use **vector similarity**.

Cosine Similarity

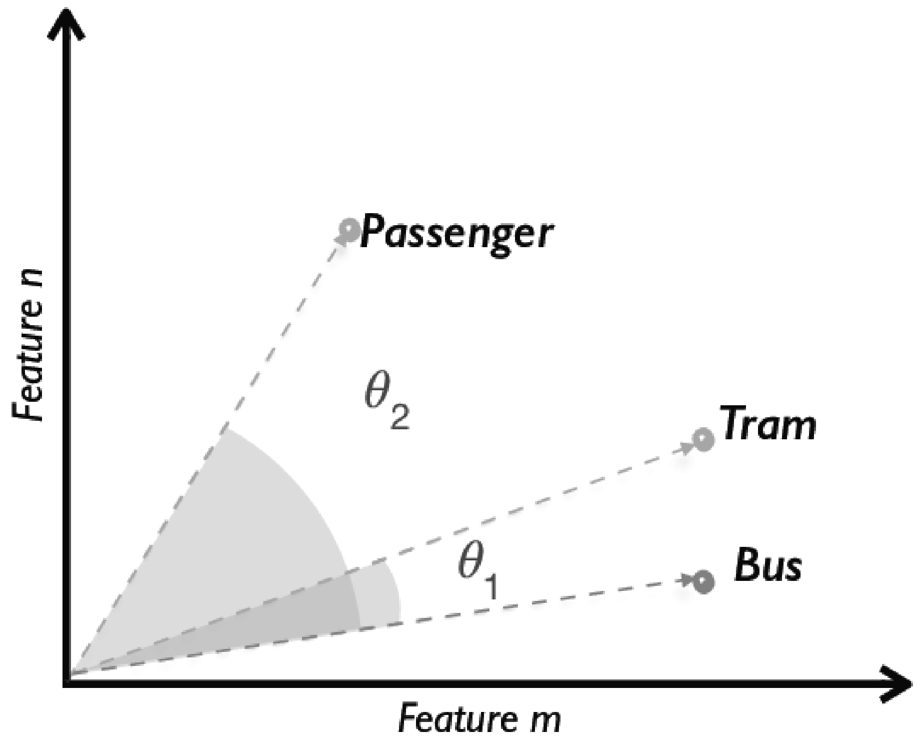
Most common metric used to compare word vectors:

$$\text{cosine\_similarity}(\vec{A}, \vec{B}) = \frac{\vec{A} \cdot \vec{B}}{\|\vec{A}\| \|\vec{B}\|}$$

- Measures the **angle** between two vectors (not their magnitude).
- Ranges from **-1 to 1**:
  - 1 → Same direction (very similar)
  - 0 → Orthogonal (unrelated)
  - -1 → Opposite directions (very different)

Intuition

- Vectors for `"king"` and `"queen"` will have high cosine similarity.
- Vectors for `"apple"` and `"keyboard"` will have low similarity.



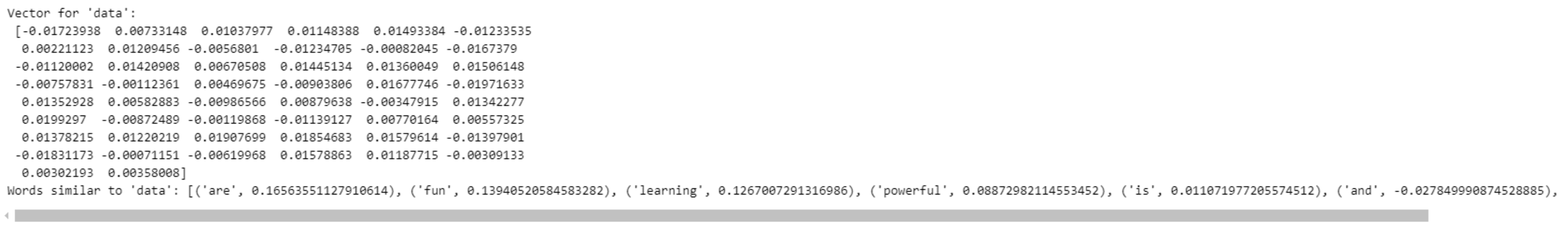
Experimenting with Fake Embeddings

```python
import numpy as np

from sklearn.metrics.pairwise import cosine_similarity

# Fake word vectors (3D for simplicity)

word_vectors = {

    "king": np.array([0.8, 0.65, 0.1]),

    "queen": np.array([0.78, 0.66, 0.12]),

    "man": np.array([0.9, 0.1, 0.1]),

    "woman": np.array([0.88, 0.12, 0.12]),

    "apple": np.array([0.1, 0.8, 0.9]),

}

def similarity(w1, w2):

   return cosine_similarity([word_vectors[w1]], [word_vectors[w2]])[0][0]


print("Similarity(king, queen):", similarity("king", "queen"))

print("Similarity(man, woman):", similarity("man", "woman"))

print("Similarity(king, apple):", similarity("king", "apple"))
```

Open in Colab

Learn Embeddings From Scratch

```python
from gensim.models import Word2Vec
# Sample corpus
sentences = [['data', 'science', 'is', 'fun'],
    ['machine', 'learning', 'is', 'powerful'],
    ['data', 'and', 'learning', 'are', 'related']]
# Train the model
model = Word2Vec(sentences, vector_size=50, window=2, min_count=1, workers=2)
# Access the embedding for a word
print("Vector for 'data':\n", model.wv['data’])
# Find similar words
print("Words similar to 'data':", model.wv.most_similar('data'))
```



```
Vector for 'data':
 [-0.01723938  0.00733148  0.01037977  0.01148388  0.01493384 -0.01233535
  0.00221123  0.01209456 -0.0056801  -0.01234705 -0.00082045 -0.0167379
 -0.01120002  0.01420908  0.00670508  0.01445134  0.01360049  0.01506148
 -0.00757831 -0.00112361  0.00469675 -0.00903806  0.01677746 -0.01971633
  0.01352928  0.00582883 -0.00986566  0.00879638 -0.00347915  0.01342277
  0.0199297  -0.00872489 -0.00119868 -0.01139127  0.00770164  0.00557325
  0.01378215  0.01220219  0.01907699  0.01854683  0.01579614 -0.01397901
 -0.01831173 -0.00071151 -0.00619968  0.01578863  0.01187715 -0.00309133
  0.00302193  0.00358008]
Words similar to 'data': [('are', 0.1656355112791061 4), ('fun', 0.13940520584583282), ('learning', 0.1267007291316986), ('powerful', 0.08872982114553452), ('is', 0.011071977205574512), ('and', -0.027849990874528885),
```

Open in Colab

Use Pre-Trained Embeddings

Gensim

- Gensim is a powerful open-source Python library designed specifically for unsupervised topic modeling and natural language processing tasks, with a strong focus on working with large corpora.
- It excels in handling word embeddings and semantic similarity, offering efficient implementations of models like Word2Vec, FastText, and Doc2Vec.
- Gensim is known for its memory-efficient, streaming-based approach, which allows it to process text data without loading everything into memory.
- This makes it especially useful for working with real-world, large-scale text data.

```python
import gensim.downloader as api
from gensim.models import Word2Vec

# Load pre-trained Word2Vec model
word2vec_model = api.load("word2vec-google-news-300")

# Find similar words
similar_words = word2vec_model.most_similar('computer', topn=5)
print("Words similar to 'computer':", similar_words)

# Word analogies
result = word2vec_model.most_similar(positive=['woman', 'king'],
    negative=['man'], topn=1)
print("king - man + woman =", result)

# Train your own Word2Vec model
sentences = [["cat", "say", "meow"], ["dog", "say", "woof"]]
model = Word2Vec(sentences, vector_size=100, window=5, min_count=1,
    workers=4)

# Get vector for a word
cat_vector = model.wv['cat']
print("Vector for 'cat':", cat_vector[:5])  # Show first 5 dimensions
```

Open in Colab

Word Embeddings Example using Spacy Library

spaCy

- spaCy is a fast and robust natural language processing library for Python that provides industrial-strength tools for text preprocessing and linguistic analysis.
- It comes with pre-trained models for multiple languages and supports features like tokenization, part-of-speech tagging, named entity recognition, dependency parsing, and sentence segmentation.
- spaCy is designed for performance and ease of use in production environments and integrates well with deep learning libraries.
- While it's not primarily focused on word embeddings, it includes pre-trained word vectors and supports similarity comparisons out of the box.

- pip install spacy
- python -m spacy download en_core_web_md

```
import spacy
nlp = spacy.load("en_core_web_md")

word1 = nlp("king")
word2 = nlp("queen")
print("Similarity:", word1.similarity(word2))
```

Open in Colab

Applications of Word Embeddings in NLP

1. **Semantic Similarity**
   Measure how similar two words, phrases, or documents are by comparing their vector representations.
   Example: Identifying that "doctor" and "physician" are closely related.
2. **Text Classification**
   Used as input features for tasks like spam detection, sentiment analysis, and topic classification.
   Embeddings provide rich, dense input for machine learning models.
3. **Named Entity Recognition (NER)**
   Help identify proper nouns and classify them into categories like person, location, or organization.
   Embedding-based models improve contextual understanding of named entities.
4. **Machine Translation**
   Map words from one language to another by aligning embeddings in multilingual space.
   Improves translation accuracy by leveraging semantic proximity.
5. **Question Answering & Chatbots**
   Used to understand queries and match them with appropriate answers or responses.
   Enable bots to interpret intent and context more accurately.

- **6. Information Retrieval**
  Enhance search engines by retrieving results based on semantic meaning, not just keyword matches.
  Example: Searching for “heart attack” returns documents containing “cardiac arrest.”